|  |
| --- |
|  |
| **REPORT**  제목 : 4주차\_고급소프트웨어실습\_결과보고서 |
|  |
| |  |  |  |  | | --- | --- | --- | --- | | 제출일 | 2021. 10. 17 | 전 공 | 컴퓨터공학과 | | 과 목 | 고급소프트웨어실습 | 학 번 | 20182204 | | 담당교수 | 임인성 | 이 름 | 한찬희 | |

1. 실습 목적

Newton-Rapson, Secant, Bisection 방식으로 방정식의 해를 구해본다.

1. 실습 구현 내용

* Newton-Rapson 방법
* Secant 방법
* Bisection 방법

1. 실습 환경

Hansung TFX255S

CPU : Intel Core i5-10210U 1.6GHz (Cometlake)

RAM : 8GB PC4-21300

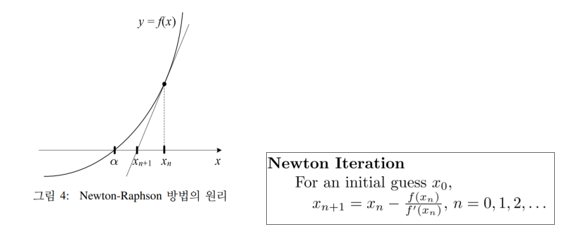
GPU : NVIDIA Geforce MX250 GDDR5 2GB

OS : Windows10 Pro

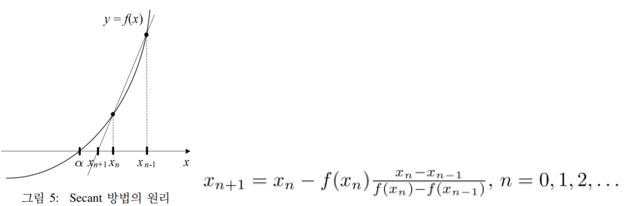
IDE : vs 2019

1. 실습 결과 및 분석

* **프로그램 설명 및 구동 방법**
  + 프로그램 기본 구성은 main.cpp 에서 function.cpp 혹은 sp\_function.cpp 에 정의된 함수를 함수 포인터 변수 \_f, \_fp에 저장하여 program1\_1, program1\_2, program1\_3, sp\_program1\_1 함수들이 각각 사용함으로 결과를 확인할 수 있다. function.cpp에는 이번 실습 1, 2, 4에 사용될 함수 3가지와 과제에서 사용될 함수 한 개가 정의되어 있다. sp\_function.cpp에는 실습 4에 사용될 반환형이 float인 함수들이 정의되어 있다. program1\_1에는 Newton-Rapson 방법의 함수가 정의되어 있다. program1\_2에는 Scant 방법의 함수가 정의되어 있다. program1\_3에는 과제에서 사용될 Bisection 방법이 정의되어 있다. sp\_program1\_1 에는 program1\_1과 동일한 Newton-Rapson 방법이 정의되어 있으나 float로 변수와 반환형이 정의되어 있다.
* **실습 문제 1-1** 
  + 실습 1-1에서는 Newton-Raphson 방법과 Secant 방법은 다음과 같은 원리로 정의되어 있다.



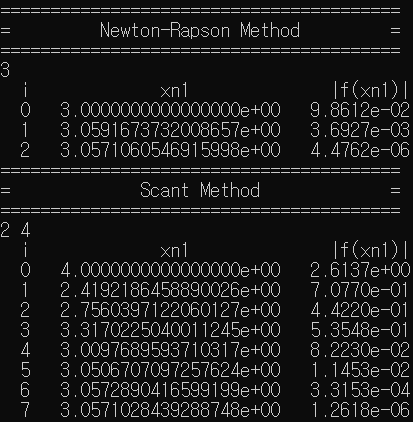
Newton-Rapson 방법의 원리와 수식



Secant 방법의 원리와 수식

모든 실습 문제에 대해서 DELTA는 0.000001, EPSILON은 0.00001, Nmax는 50으로 설정하였다.

첫번째 실습 문제에서 사용한 방정식은 아래와 같으며 Newton-Rapson은 3.0 Secant은 2.0, 4.0 값을 대입했다.



* + 프로그램이 원하는 근을 정확히 찾고 있는지 분석하라. 과연 근이 맞는지 어떻게 확인할 수 있을까?

Newton-Rapson 방법과 Scant 방법이 회차를 반복함에 따라서 |f(xn1)|의 값이 점차 0에 수렴해가는 것을 확인할 수 있다. 이를 통해 프로그램이 원하는 근을 정확히 찾고 있다는 것을 확인할 수 있다.

* + 각 방법의 근의 수렴 속도가 과연 앞에서 설명한 속도로 수렴하는지 비교 분석한 후 그 결과를 보고서에 기술하라

이론상에 따르면 Newton-Rapson 방법의 절대 오차는 다음의 공식을 따른다고 확인할 수 있다.

f1(x) Newton-Raphson 실행 결과에 따른 절대 오차

|  |  |
| --- | --- |
| 실행횟수 | 오차값 |
| 0 | 0.0571035 |
| 1 | 0.0020638 (여기서 c는 약 0.6329로 나오게 되었다.) |
| 2 | 0.0000025 (여기서 c는 약 0.5849로 나오게 되었다.) |

i번째에 따른 절대오차를 이용해 계산한 c의 값이 완벽히 같진 않지만 대략적으로 비슷하게 나와 이론에 어느정도 일치하는 것을 알 수 있었다.

이론상 Secant 방법의 경우 절대 오차가 대략적으로 다음과 같이 감소하게 된다.

f1(x) Secant 실행결과에 따른 절대 오차

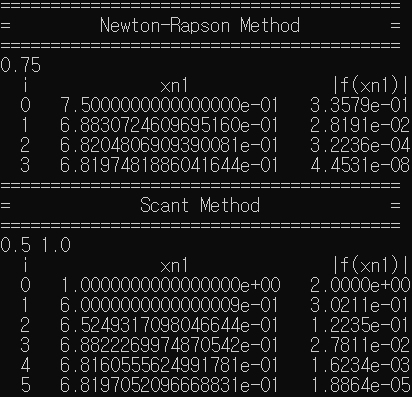
|  |  |
| --- | --- |
| 실행횟수 | 오차값 |
| 0 | 0.9428965 |
| 1 | 0.6378849 (c ≈ 0.701633) |
| 2 | 0.3010638 (c ≈ 0.701633) |
| 3 | 0.259919 (c ≈ 1.181723) |
| 4 | 0.0473364 (c ≈ 0.419911) |
| 5 | 0.0064328 (c ≈ 0.900707) |
| 6 | 0.0001855 (c ≈ 0.658773) |
| 7 | 0.0000006561 (c ≈ 0.7282037) |

이론적인 측면에서 확인해보았을 때 오차 감소 과정을 보면 Newton-Raphson이 Secant에 비해 보다 수렴 속도가 빠른 것을 확인할 수 있었다.

실습에서의 수렴 속도는 원하는 정밀도를 가지는 근 추정값을 구하는데 필요한 반복문의 수행 횟수를 의미한다고 가정한다. Newtown-Rapson 3회 Secant는 8회정도 연산이 수행된 것을 확인하여 Newton-Rapson 방법이 Secant에 비해 빠른 게 근에 수렴하는 것을 확인할 수 있었다.

* + 제시된 초기값 외의 임의의 초기값들을 사용하여 자신이 작성한 프로그램을 수행한 후, 이 두 방법이 항상 임의의 초기값에 대해 빠르게 수렴하는지 보고서에 기술하라

두번째 함수의 결과



임의의 초기값 0.75와 0.5, 1.0 을 사용하여 해를 구한 모습을 확인할 수 있다.

f1(x) Newton-Raphson 실행 결과에 따른 절대 오차

|  |  |
| --- | --- |
| 실행횟수 | 오차값 |
| 0 | 0.0571035 |
| 1 | 0.0020638 (여기서 c는 약 0.6329로 나오게 되었다.) |
| 2 | 0.0000025 (여기서 c는 약 0.5849로 나오게 되었다.) |

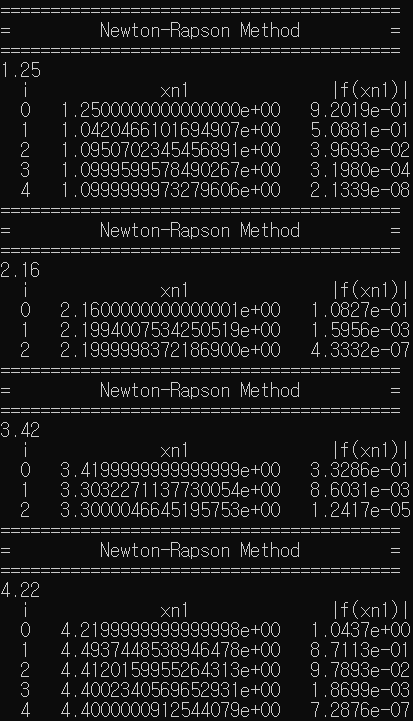
앞서 확인한 Newton-Raphson 방법에 따르면 오차값은 제시된 함수의 형태로 줄곧 감소하게 된다.

f1(x) Secant 실행결과에 따른 절대 오차

|  |  |
| --- | --- |
| 실행횟수 | 오차값 |
| 0 | 0.9428965 |
| 1 | 0.6378849 (c ≈ 0.701633) |
| 2 | 0.3010638 (c ≈ 0.701633) |
| 3 | 0.259919 (c ≈ 1.181723) |
| 4 | 0.0473364 (c ≈ 0.419911) |
| 5 | 0.0064328 (c ≈ 0.900707) |
| 6 | 0.0001855 (c ≈ 0.658773) |
| 7 | 0.0000006561 (c ≈ 0.7282037) |

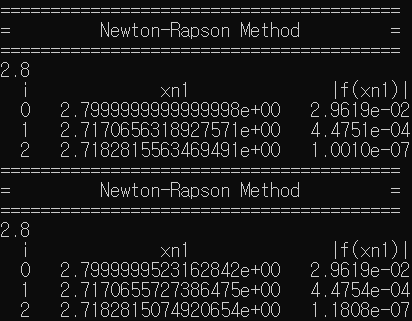
Secant 방법 역시 공식에 따라 오차값이 점차 감소하는 것을 확인할 수 있다. 이를 통해 두 방법은 임의의 상수값이 입력되더라도 Nmax 이내의 시간안에 원하는 해를 구할 수 있다고 판단할 수 있다.

* **실습 문제 1-2**



이 방정식은 4차 방정식으로 서로 다른 4개의 근을 가지고 있다. 따라서 각 함수롤 4번 실행하여 4개의 해를 모두 알아낼 수 있어야 했다. 이때 사용된 값은 각 구간의 중간값(1.25, 2.16, 3.42, 4.22)로 결정하여 결과를 구했다.

* **실습 문제 1-4**



위쪽은 double-precision 버전으로 계산한 반복문의 결과이고 아래쪽은 single-precision 버전으로 나뉘어 구한 결과이다. 초기값은 모두 2.8로 설정하였다.

* + 부동 소수점 연산의 정밀도가 다른 두 방법이 구한 근의 값이 정확한 근 e와 비교하여 어떤 차이가 있는지, 자신이 알아낸 사실을 상세히 기술하라.

몇번의 시도 끝에 확인할 수 있는 바는 single-precision에 비해 double-precision으로 구한값의 오차가 더 적은 것을 확인할 수 있었다. 이는 double과 float 자료형의 크기 차이로 표현할 수 있는 데이터의 범위 차이로 인해 발생한 차이점이라고 확인할 수 있다.

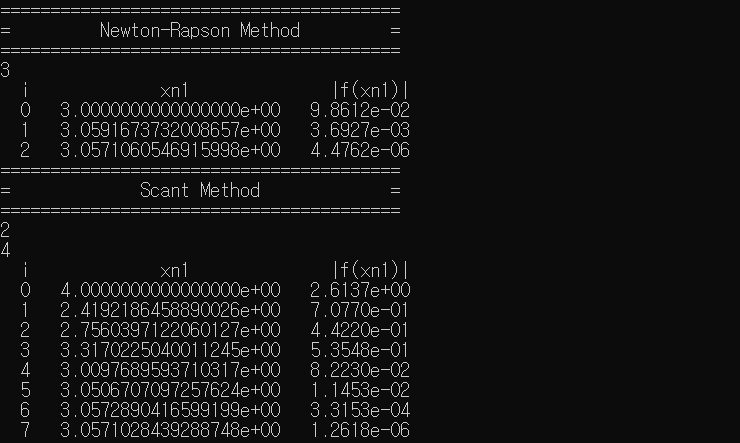
1. 과제

* **과제 1 – Bisection**

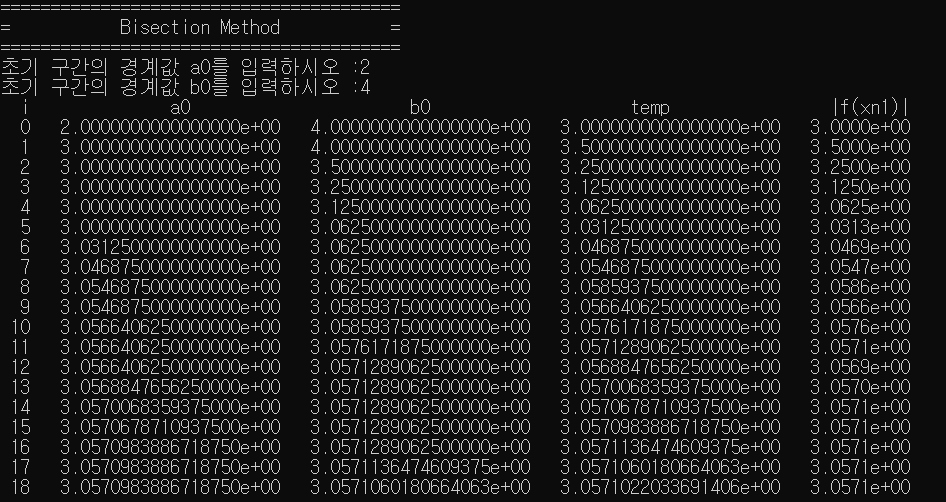
Bisection은 비선형 연립 방정식의 근을 구하기 위해 초기 경계값 a0, b0를 입력해야 한다. 입력된 값을 기준으로 중앙값 xn을 구하게 되는데 이때의 중앙값을 넣은 함수 f(xn)이 0이거나 델타값보다 작거나 b0-a0의 값이 앱실론보다 작다면 값을 찾아낸것으로 판단한다.

만약 xn의 값이 원하는 결과가 아니라면 f(a0) \* f(xn)의 결과가 0 미만인지 확인한다 만약 0 미만이라면 a0와 xn 사이에 해가 있다는 뜻이기 때문에 b0의 범위를 xn으로 바꾼다. 그 반대로면 a0를 xn의 값으로 바꾼다.

* + 프로그램이 완성되면 f1(x), f2(x), f3(x)에 대해 적절한 초기 구간을 사용하여 올바르게 근에 수렴하는지에 대한 분석 내용을 보고서에 기술하라

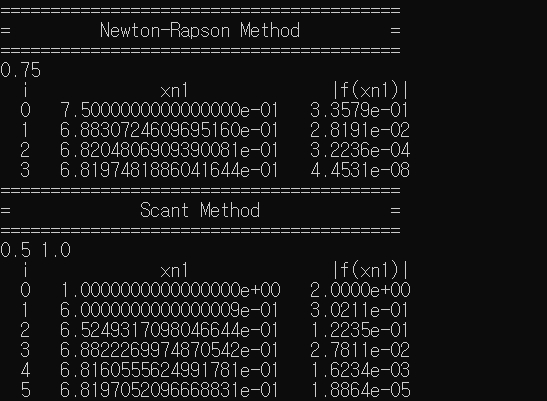


f1(x)의 Newton-Rapson과 Scant 방법으로 수행한 결과

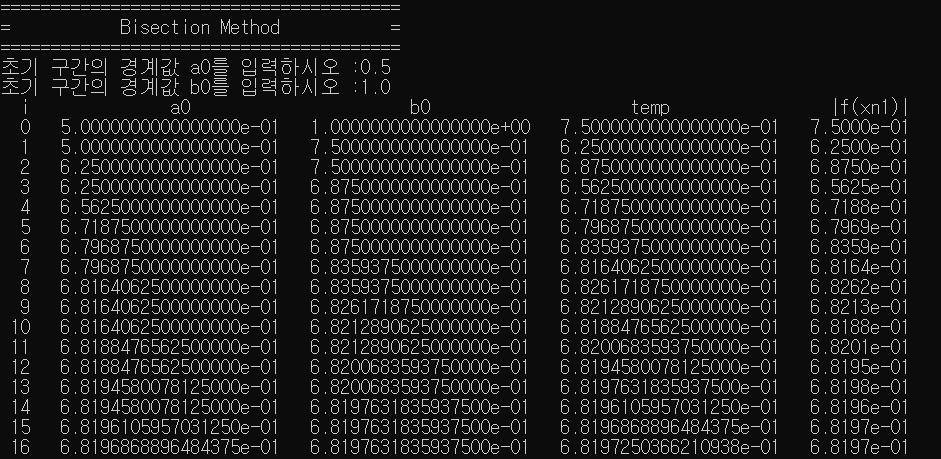


f1(x)의 Bisection으로 수행한 결과

Bisection 방법의 적절한 초기 구간 경계값을 설정하기 위해 함수에 값을 대입해 부호를 비교하여 설정하였다. 2.0과 4.0이 조건에 맞아 각각 a0, b0에 할당되었다. 세 방법의 x의 값과 |f(x)|가 0에 가까이 가는지를 확인하여 올바르게 근에 수렴하는지 확인할 수 있었다.

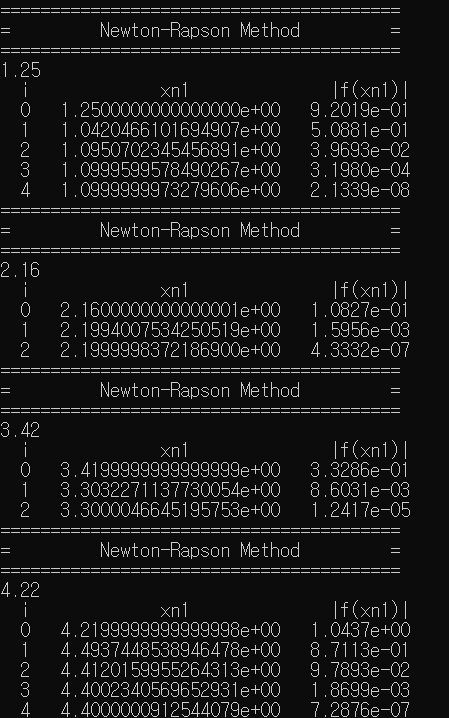


f2(x)의 Newton-Rapson Method와 Scant Method 사용결과

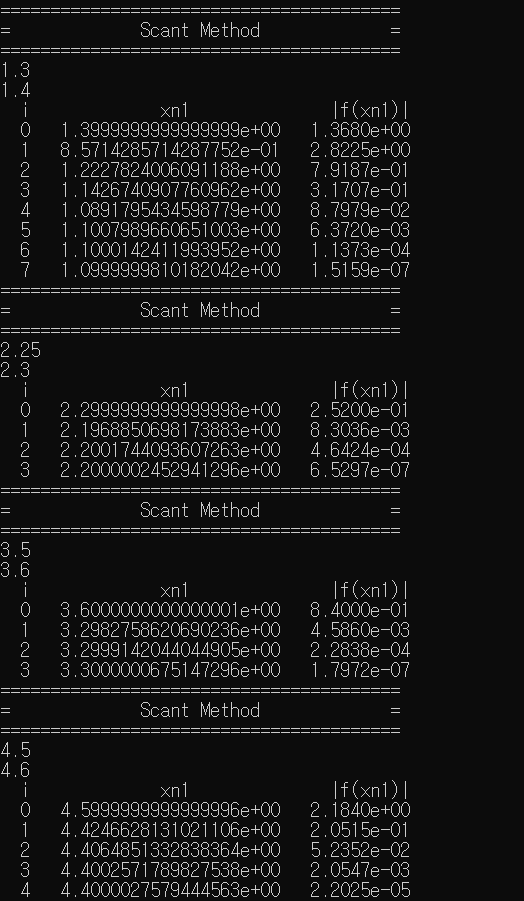


f2(x)의 Bisection Method 사용결과

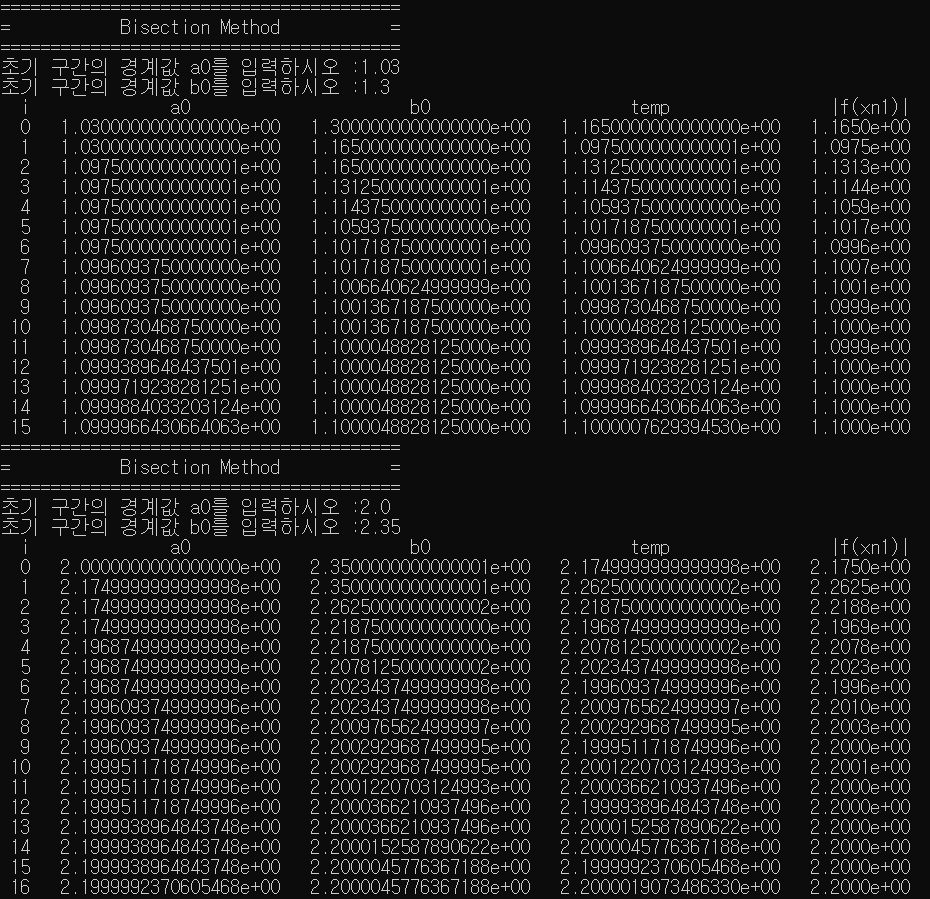
Bisection 방법의 적절한 초기 구간 경계값을 설정하기 위해 함수에 값을 대입해 부호를 비교하여 설정하였다. 0.5과 1.0이 조건에 맞아 각각 a0, b0에 할당되었다. 세 방법의 x의 값과 |f(x)|가 0에 가까이 가는지를 확인하여 올바르게 근에 수렴하는지 확인할 수 있었다.

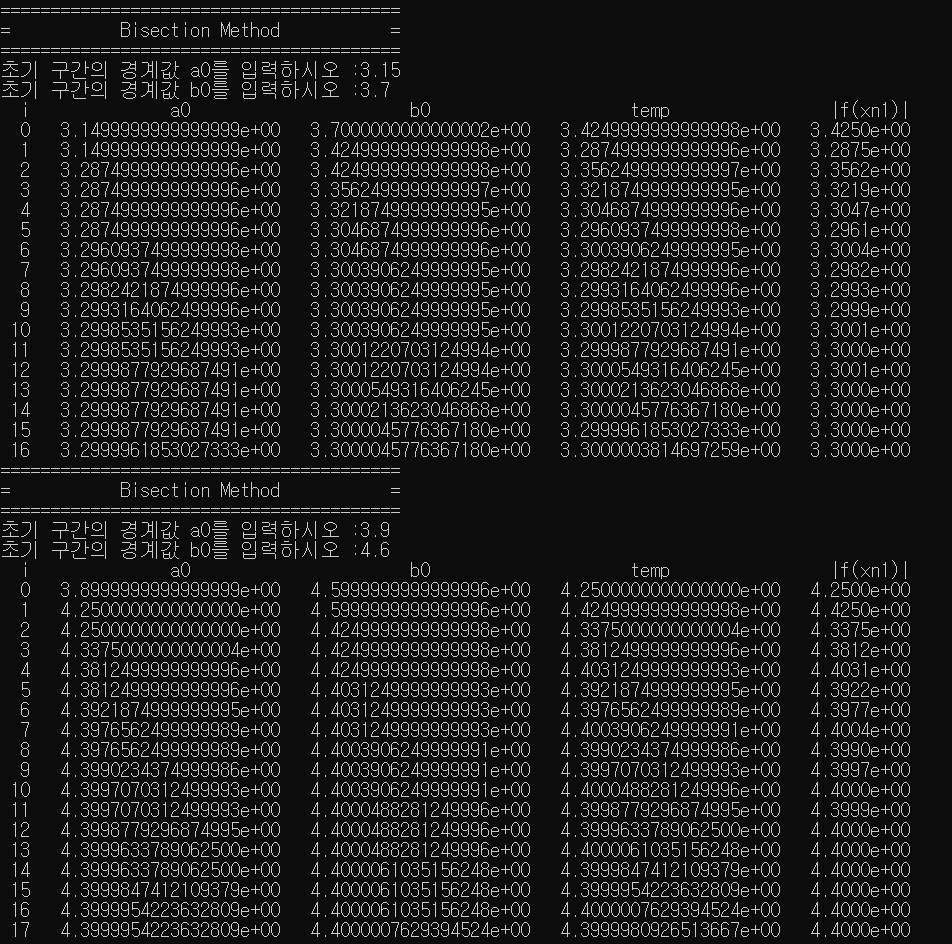


f3(x)의 Newton-Rapson Method 사용결과



f3(x)의 Scant Method 사용결과





f3(x)의 Bisection Method 사용결과

f3(x)의 근을 Secant Method와 Bisection Method로 구하는 과정은 Newton-Raphson과 같이 함수를 4번 호출하여 각 근에 대한 초기값 두 개를 콘솔로 입력받아 진행하였다. Bisection Method의 적절한 초기 구간 경계값은 근이 존재하는 범위를 참고하였고, [1.03, 1.3], [2.0,2.35], [3.15, 3.7], [3.9, 4.6]이 조건에 맞아 각각 a0과 b0에 할당하였다. 세 방법의 x의 값과 |f(x)|이 0에 가까이 가는지를 확인하여 올바르게 근에 수렴하는지 확인할 수 있었다.

* + Bisection 방법의 수렴 속도는 선형적인, 즉 형태를 보인다. 위의 세 함수에 대한 방정식을 대상으로 Newtown-Raphson 방법, Secant 방법, 그리고 Bisection 방법을 적용해보고, 과연 각 방법이 이론적인 수렴 속도를 보이는지를 분석하고 그 내용을 보고서에 기술하라
* f1(x) Bisection Method 실행 결과에 따른 절대 오차

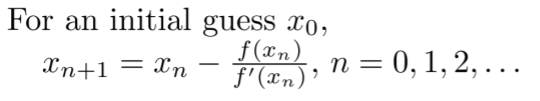
|  |  |
| --- | --- |
| 0 | 0.0571022 |
| 1 | 0.4428978 (c ≈ 7.756, 0과 1번째 오차에 대하여) |
| 2 | 0.1928978 (c ≈ 0.435535, 1번째와 2번째 오차에 대하여) |
| 3 | 0.0678978 (c ≈ 0.351988, 2번째와 3번째 오차에 대하여) |
| 4 | 0.053978 (c ≈ 0.794988, 3번째와 4번째 오차에 대하여) |
| 5 | 0.0258522 (c ≈ 0.478939, 4과 5번째 오차에 대하여) |
| 6 | 0.0102272 (c ≈ 0.395602, 5과 6번째 오차에 대하여) |
| ... | ... |

Bisection Method에서 c는 n+1번째 절대오차에서 n번째 절대오차를 나눈 값으로 계산하였다. i번째에 따른 절대오차를 이용해 계산한 c의 값이 이론처럼 완벽히 0.5가 되지는 않지만 대략적으로 비슷하게 나와 이론에 어느 정도 일치하다는 것을 알 수 있었다. f2(x), f3(x)에서도 마찬가지로 위 표와 같이 진행하여 확인하였다.

Bisection Method는 i가 19번째까지 반복문이 진행되었으므로 세 가지 방법 중 가장 속도가 오래 걸린다는 것을 확인할 수 있었다. 따라서, Newton-Raphson – Secant – Bisection 방법 순으로 속도가 빠르다는 것을 알 수 있다

* **과제 2 – Newton-Raphson 방법으로 방정식의 해를 풀이**

비선형 함수 f(x)에 대하여 f(x) = 0이란 방정식을 풀어 근을 찾기 위해 Newton-Raphson 방법을 이용하였다. 뉴턴 반복문은 다음과 같다.



초기값 x0에 대하여 위 수식을 반복하며, 반복문 종료 조건에 만족하면 반복을 종료한다. 반복문 종료 조건은 다음과 같다.

1. 현재 구한 xn+1에 대해 함수 값이 충분히 작은가? fabs(f(xn+1)) < DELTA
2. 충분히 많은 횟수만큼 반복문을 수행하였는가? i >= Nmax
3. 현재 구한 xn+1이 직전에 구한 xn에 비해 더 이상 의미 있는 진전을 하지 않는가? fabs(xn+1 – xn) < EPSILON

여기서 DELTA는 0.000001, EPSILON은 0.00001, Nmax는 50으로 설정하였다.

문제에서

l = 89, h = 49, D = 55, beta = 11.5

A = l \* sin(beta)

B = l \* cos(beta)

C = (h + 0.5 \* D) \* sin(beta) – 0.5 \* D \* tan(beta)

E = (h + 0.5 \* D) \* cos(beta) – 0.5 \* D

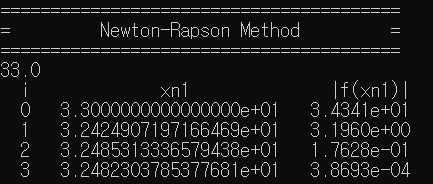
그리고 이 함수들을 이용하는 함수 방정식은 다음과 같다.(함수 이름 : \_f\_vehicle)

f(x) = A \* sin(x) \* cos(x) + B \* sin(x) \* sin(x) – C \* cos(x) – E \* sin(x)

또한 f(x)의 도함수 f’(x)는 다음과 같다. (함수 이름 : \_fp\_vehicle)

f’(x) = A \* (cos(x) \* cos(x) – sin(x) \* sin(x)) + B \* 2 \* sin(x) \* cos(x) + C \* sin(x) – E \* cos(x)

main.cpp에서 \_f는 \_f\_vehicle을, \_fp는 \_fp\_vehicle를 할당하여 Newton-Raphson 방법을 구현한 함수 program1\_1(fp)를 호출하여 계산을 진행하였다. 초기값 x0은 33으로 설정하였다. 반복 결과는 다음과 같다.



1. 결론

Newton-Rapson 방법과 Secant 함수를 배우면서 우리가 일상 생화에서 사용하는 GPS가 실재로 어떻게 우리의 위치를 확인하는지에 대해 대략적으로 확인할 수 있었다. 우리가 사용하는 네비게이션은 이것에 비하여 보다 정교하고 복잡한 시스템으로 구하여 졌기에 그 내부에 어떠한 함수가 존재하는지 호기심이 생기었다. 또한 float와 double의 오차가 생각이상으로 크다는 점을 확인하여 앞으로의 코딩에 있어서도 자료형의 중요성을 일깨울 수 있었던 순간이라고 생각한다.